RuntimeVerification

Klaus Havelund

Kestrel Technology
NASA Ames Research Center
California, USA

Where am I Sitting?

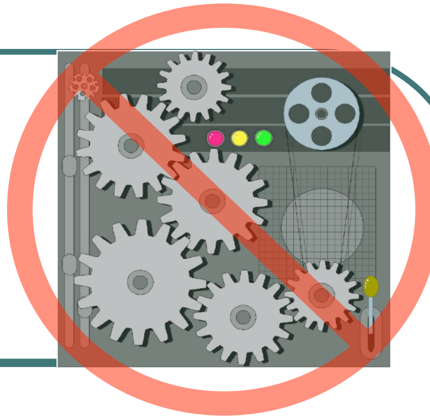
- NASA
  - JPL, California, Los Angeles: unmanned deep space
  - Houston, Texas: manned missions
  - Kennedy, Florida: launches
  - ….
  - NASA Ames, California, Mountain View:
    <u>Computer Science:</u>
    - Computational Sciences Division: 300 researchers
      - Automated Software Engineering Group
        - **Verification and Testing: 10 people**
        - Program Synthesis: 10 people
      - Planning and Scheduling
      - …
    - Super computing
    - …

The Team

- Cyrille Artho (ETH, Zurich, CH)
- Howard Barringer (U. Manchester, UK)
- Saddek Bensalem (Verimag, Grenoble, F)
- Allen Goldberg(KT/NASA Ames, USA)
- Klaus Havelund (KT/NASA Ames, USA)
- Koushik Sen (Univ. Illinois, USA)


MISSION : MARS 2016
par Pierre Boyer http://cyberd.ctw.net

NASA Increasingly Relies on Software



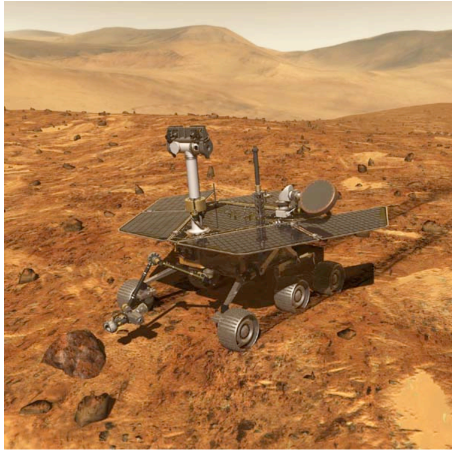
- Systems must support **remote exploration**
- Systems must be **more autonomous**
- Systems must do **more complex tasks**

Not only HW

When people think of space, they think of rocket plumes and the Space Shuttle, but the future of space is information technology…

Daniel S. Goldin,

Previous NASA Administrator



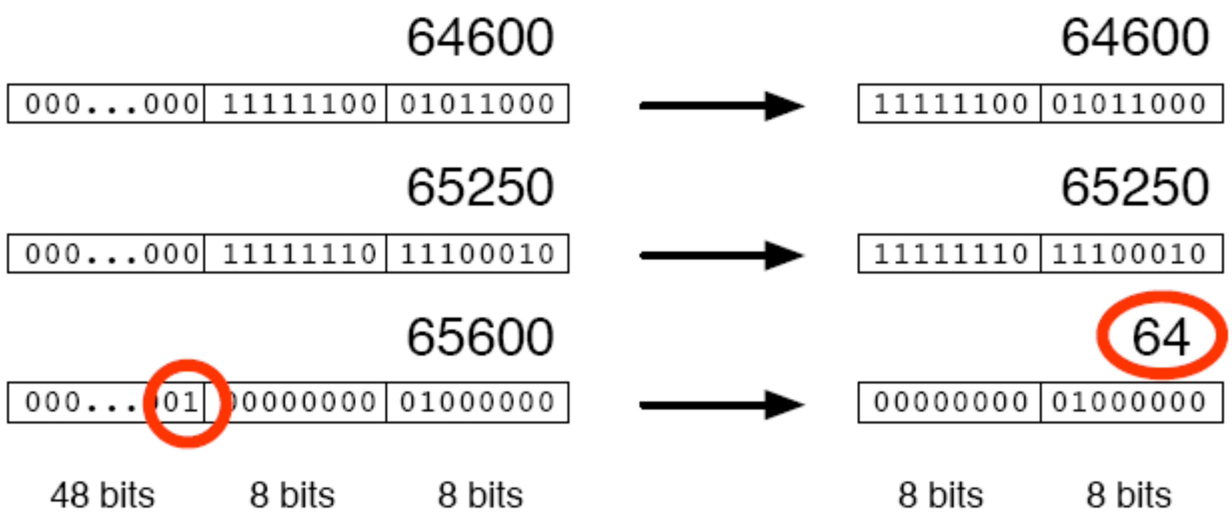
NASA gets Excited when it goes well

NASA
Kestrel Tech.

However, errors sometimes occur

Ariane 5, 1996 - Lost

Ariane:
Float to Integer conversion

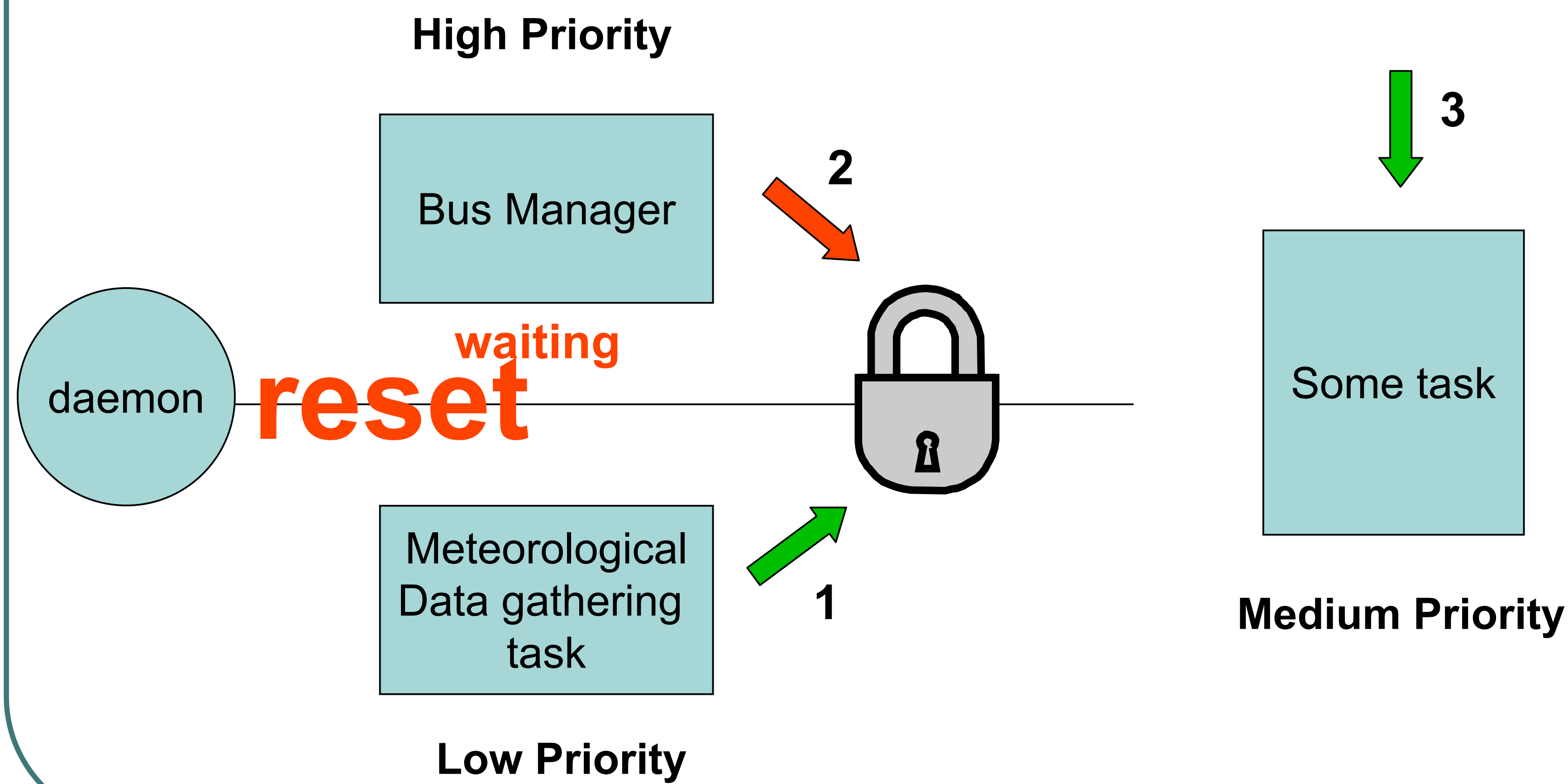


Data conversion of a too large number. 64 bit floating point number relating to the horizontal velocity of the rocket with respect to the platform was converted to a 16 bit signed integer. Number was larger than 65,536.

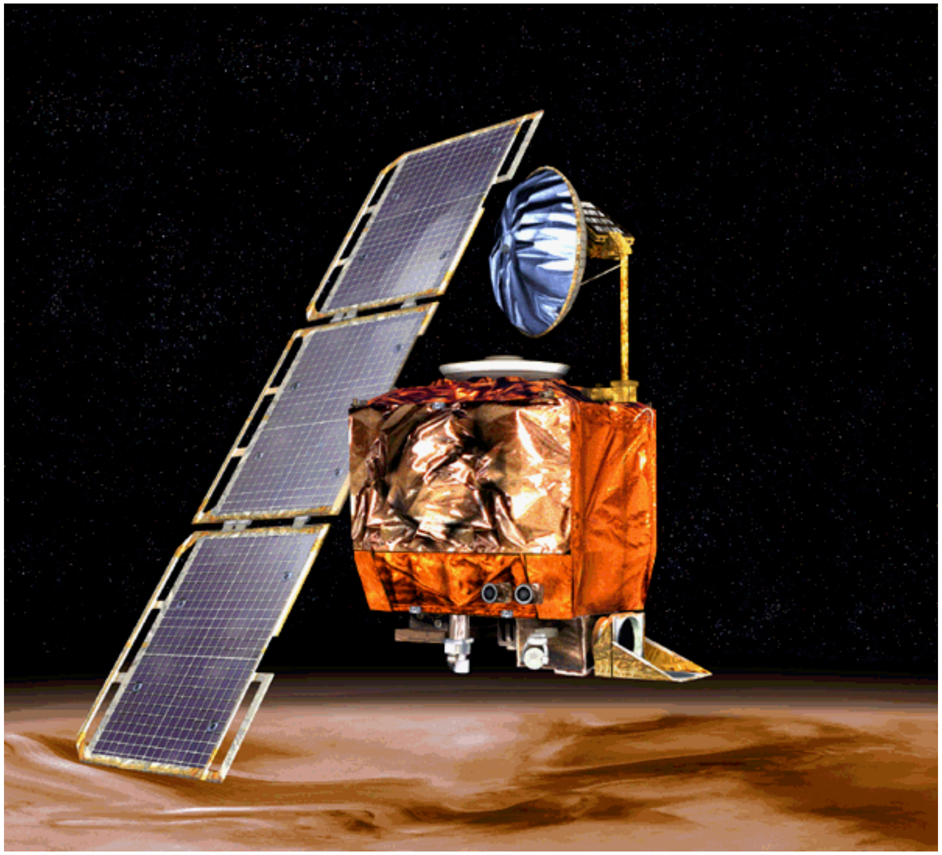
Due to higher horizontal velocity than in Ariane 4.

Mars PathFinder, 1997

Mars PathFinder: Priority Inversion Problem



Mars Climate Orbiter, 1999



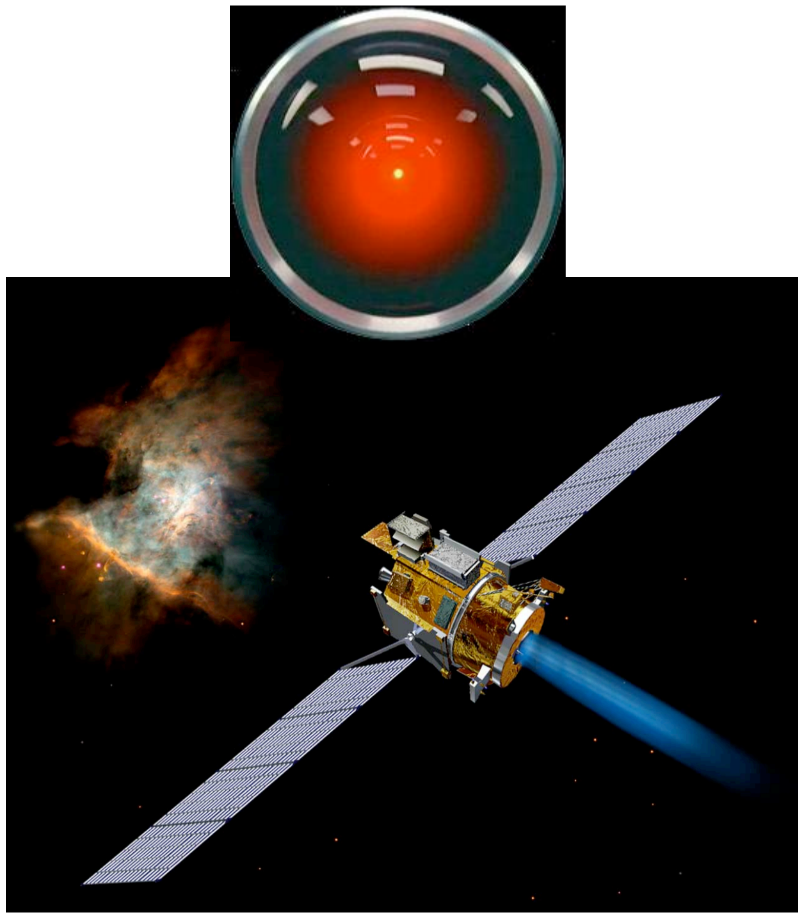
Mars Climate Orbiter: Unit error

Pounds, inches, …

Kilos, centimeters

"We had planned to approach the planet at an altitude of about 150 kilometers (93 miles). We thought we were doing that, but upon review of the last six to eight hours of data leading up to arrival, we saw indications that the actual approach altitude had been much lower. It appears that the actual altitude was about 60 kilometers (37 miles)".

Deep Space 1, 1999



Kestrel Tech.

Deep-Space 1

```
while(true){
   action();
   if(!newEvents())
      wait();
   handleEvents();
}
```

New event

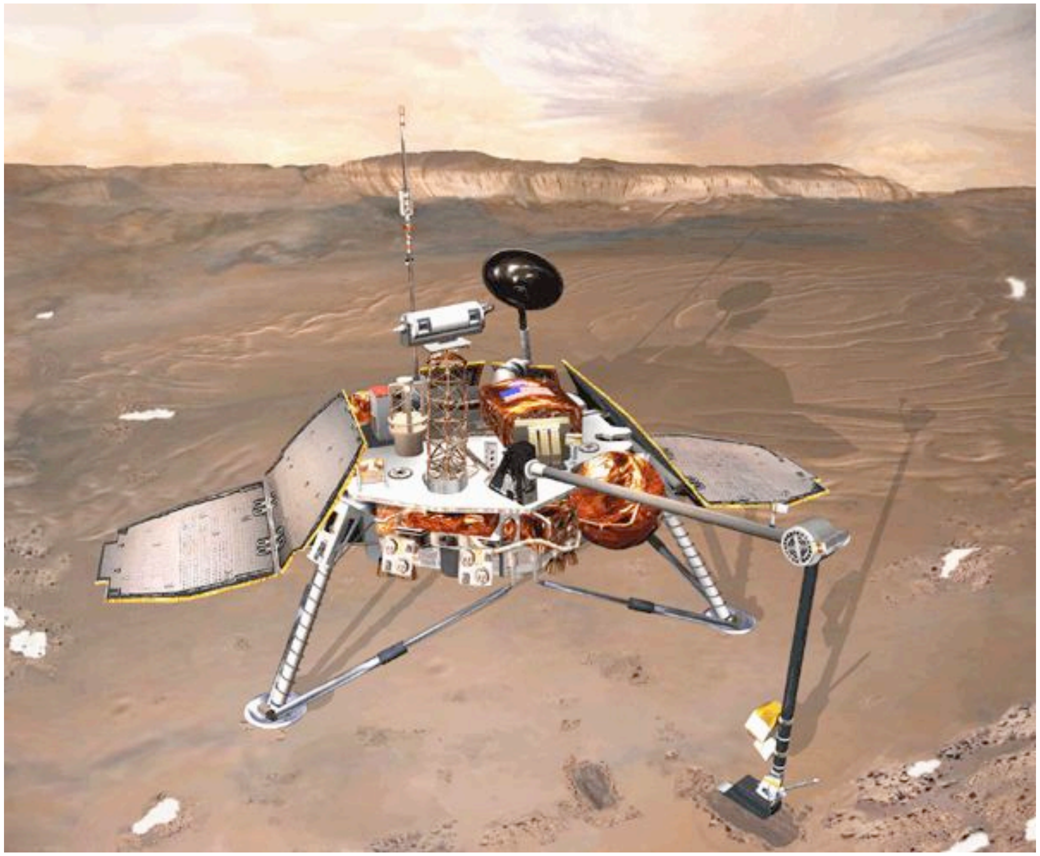
deadlock

Deep-Space1: We found error before flight

- Using the SPIN model checker
- In spacecraft operating system
- Code was corrected, but error later re-introduced in a different sub-system
- Error was located after 5 hours
- Was not fixed since modifying code could cause new errors, and …

  it was not likely to re-occur
- Shows how hard these errors are to find.

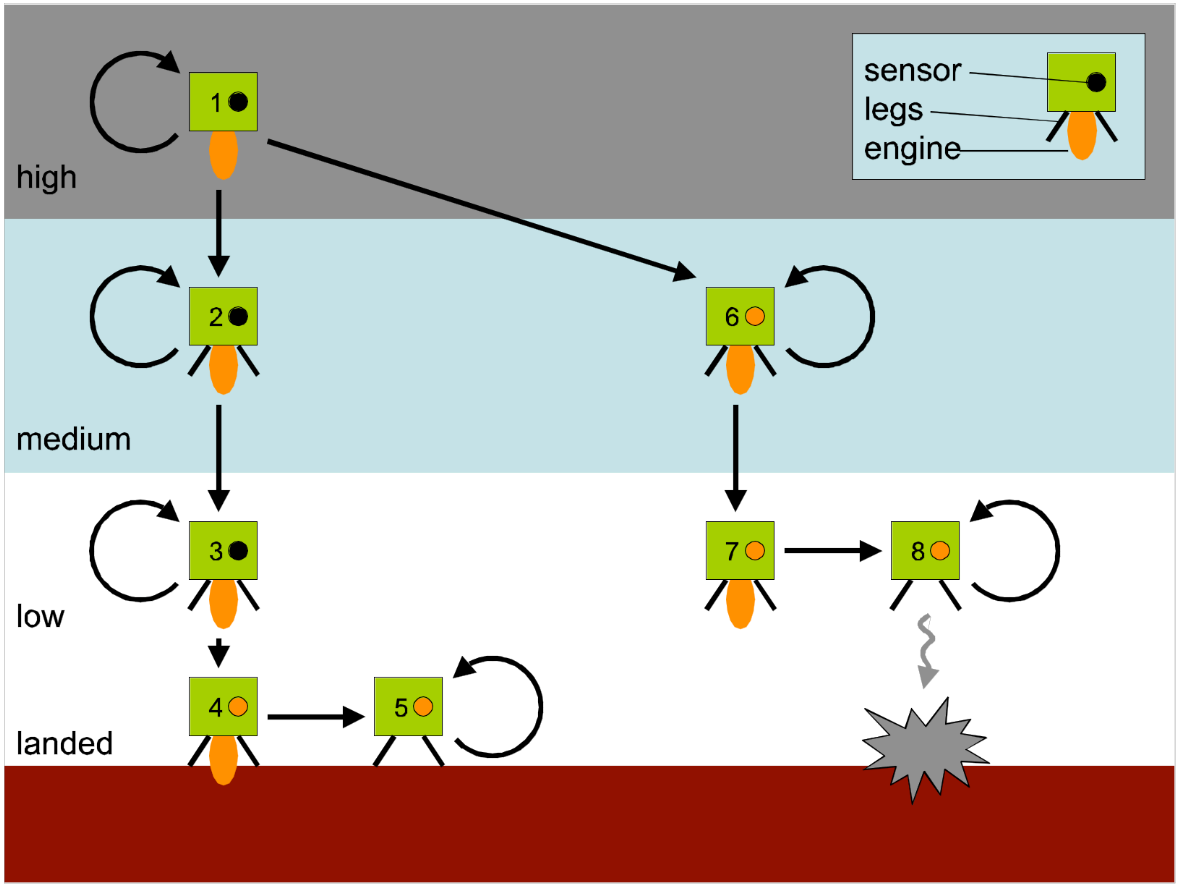
Mars Polar Lander, 1999



Mars Polar Lander: Landing Sensor activated too early

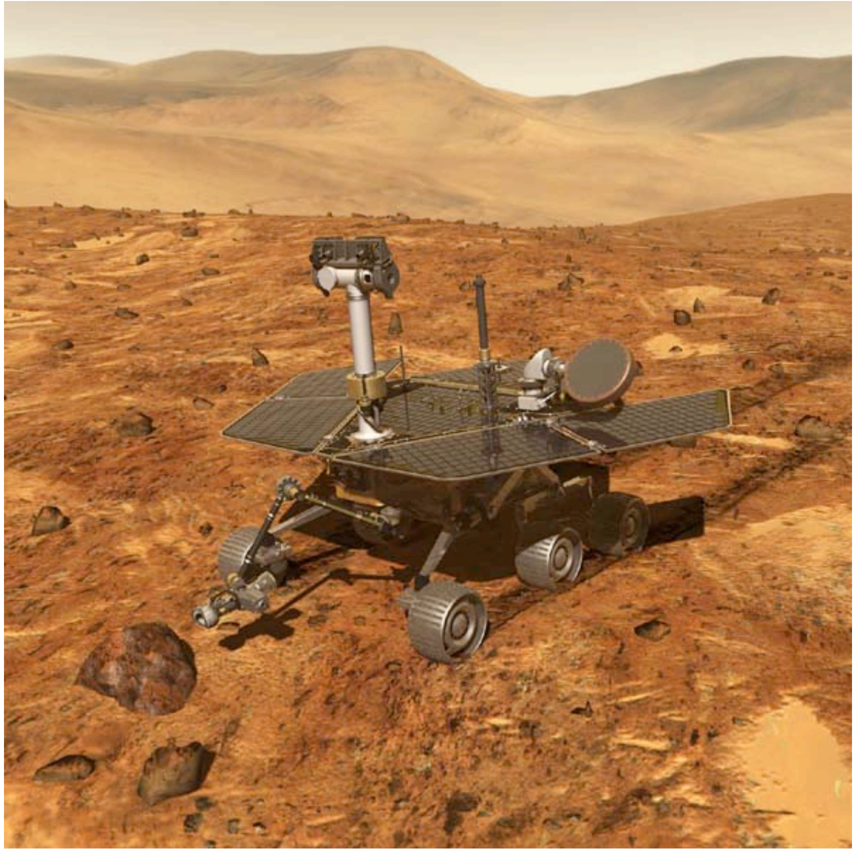
- Normally, the shake of a touch down would signal that engines should be shut down.
- However, shake of legs opening could cause the same effect in some cases. It was known.
- System was designed to ignore such shakes above 40 feet where legs were to open.
- System above 40 feet correctly ignored landed-flag, but flag was not reset to false, and triggered engine shut-off as soon as 40 feet were reached.

Mars Polar Lander: Imagined Scenario

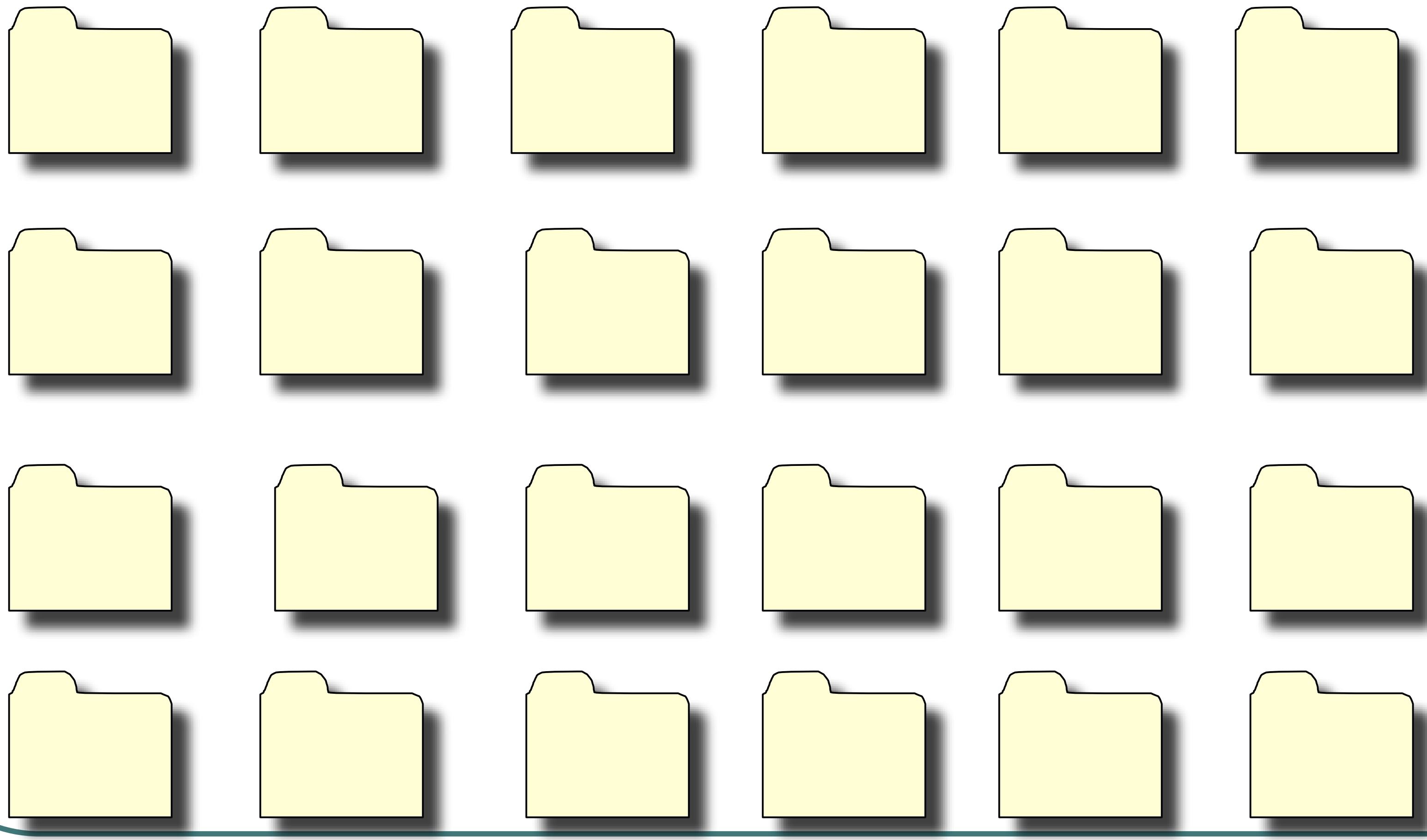


40 feet ~ 13 meters

Spirit Mars Rover, 2004



Spirit Mars Rover:
Too many files allocated



Some Observations

- Software applications for space missions have grown from a few thousands of lines of code in the late seventies to hundreds of thousands of lines of code for today's missions.
- At the current rate, the code size for controlling spacecraft *doubles* in size every four years.
- Software should be expected to contain between 1 and 10 defects per 1,000 lines of code excluding comments and blank lines.
- We are talking about hundreds of errors on current missions, and more to come.

Also Complexity grows

But, software is not just rapidly growing in size; it is also rapidly growing in complexity. Virtually all current missions use multi-threaded software designs: running up to 50 threads executing concurrently and requiring synchronization of potentially conflicting tasks.

What can We Do?

prevent, *detect*, and *control*

What can We Do?

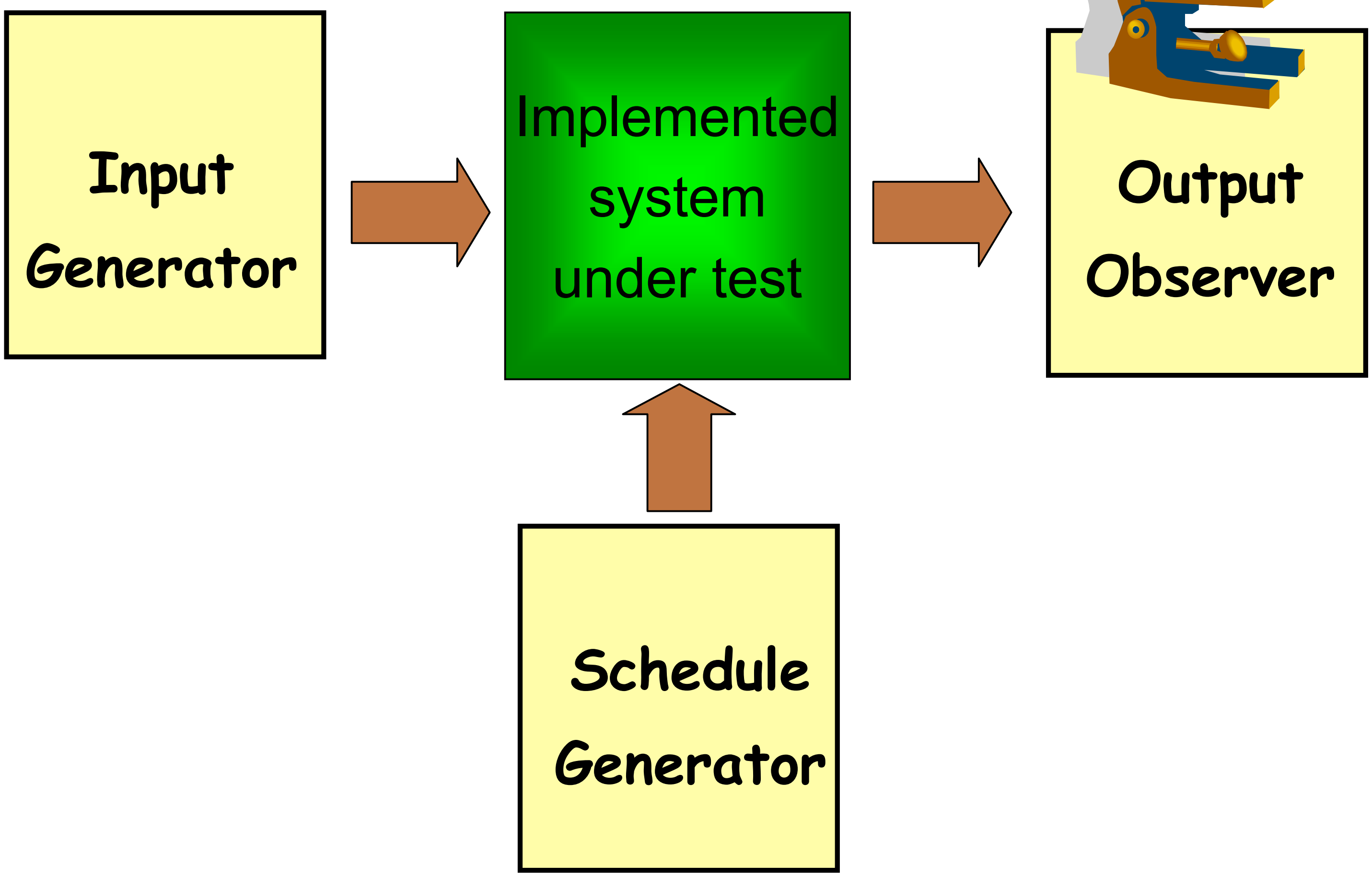
Solid formal designs
Safe programming languages

prevent, *detect*, and *control*

Test
Analysis

Fault containment

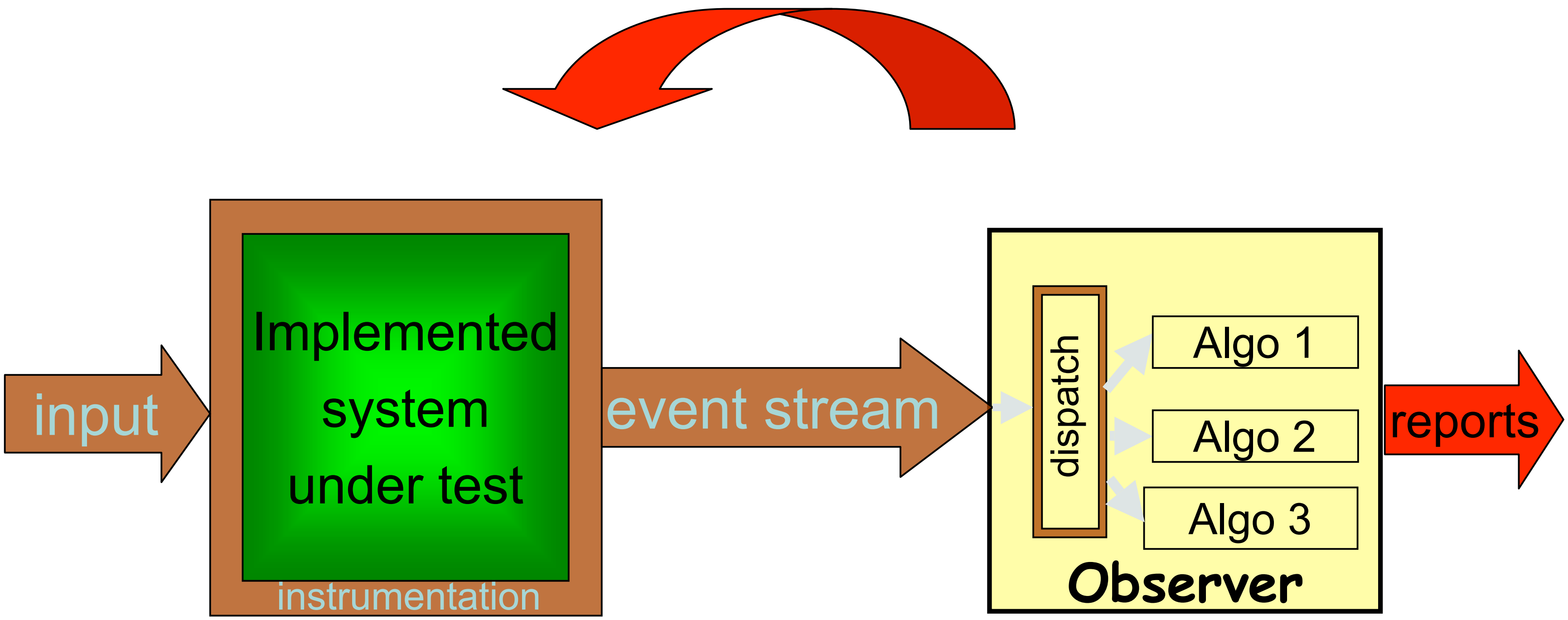
Testing



Input Generator
Implemented system under test
Output Observer
Schedule Generator

Runtime Verification

Fault containment



What is An Event Stream?

```
while(true){
  if(x>0)lock(L);
  x = shared;
  shared = f(x);
  release(L);
}
```

What is An Event Stream?

```
while(true){
  if(x>0){
    lock(L);
    logLock(t,L);
  }
  x = shared;
                logWr('x',x);
  shared = f(x);
  logWr('shared',shared);
  release(L);
  logRelease(t,L);
}
```

Instrument program.
For example using
Aspect Oriented Programming

execute

```
Trace:
,L)
lock(t2,L)
x=12
shared=24
release(t2,L)
lock(t2,L)
x=24
shared=48
release(t2,L)
lock(t2,L)
x=50
shared=100
release(t2,L)
lock(t2,L)
x=100
```

monitor

Runtime Verification Algorithms

- Requirement monitoring
  - The Eagle Temporal logic
- Concurrency Analysis
  - Deadlock analysis
  - Data race analysis
    - Low level data races
    - High level data races
    - Data flow races

Runtime Verification Algorithms

- Requirement monitoring
  - **The Eagle Temporal logic**
- Concurrency Analysis
  - Deadlock analysis
  - Data race analysis
    - Low level data races
    - High level data races
    - Data flow races

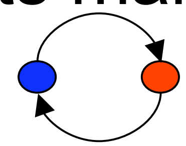
Requirement Monitoring
The Eagle Temporal Logic

Allows easy specification of properties of an execution/log file:

- **Safety properties**:
  "x is always positive".
- **Liveness properties**:
  "a turn signal is followed by a turn within 10 seconds".
- **Past and future**:
  "When a turn occurs, a turn command has been emitted before".

Requirement Specification: So many logics, notations, languages ...

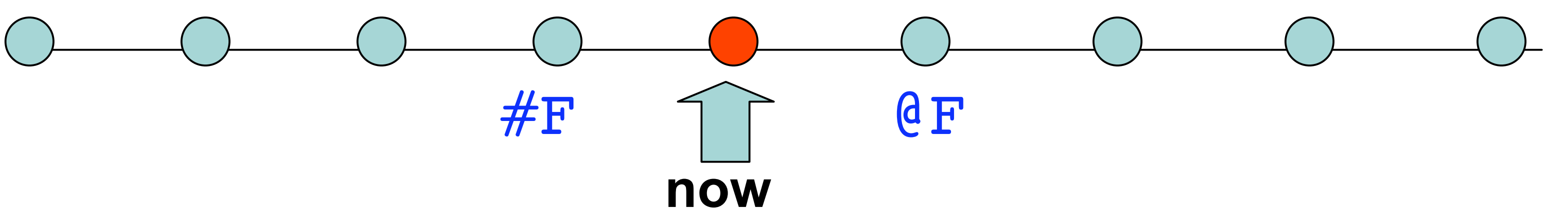
General specification language suitable for monitoring? Supports many styles:

- state machines 
- temporal logic: $\Diamond x>0$ (eventually x>0) future+past
- regular expressions: login+ use* logout
- real-time properties: $\Diamond[10]x>0$ 
- properties about data values over time: $\Box(\mathrm{login}(u) \rightarrow \Diamond \mathrm{logout}(u))$

Eagle's Core Concepts

- Three temporal connectives:
  - **Next:** `@F`
  - **Previous:** `#F`
  - **Concatenation**: `F1;F2`
- Recursive parameterized rules over trace

```
Always(Term t) = t /\ @Always(t) .
```



Basic LTL Combinators

library

```
// Future time combinators

max Always(Term t) = t /\ @ Always(t) .
min Eventually(Term t) = t \/ @ Eventually(t) .
min Until(Term t1,Term t2) =
      t2 \/ (t1 /\ @ Until(t1,t2)) .

// Past time combinators

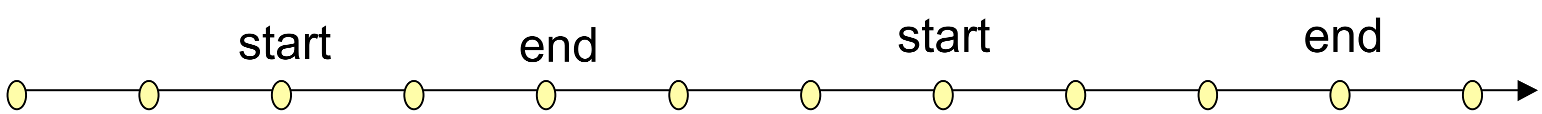
max Sofar(Term t) = t /\ # Sofar(t) .
min Previously(Term t) = t \/ # Previously(t) .
min Since(Term t1,Term t2) =
      t2 \/ (t1 /\ # Since(t1,t2)) .
```

Example

Property:
Every start is followed by an end, and vice versa.

```
mon M1 = Always(start() -> Eventually(end()))

mon M2 = Always(end() -> Previously(start()))
```



Data Bindings

Property:
when x>0 then
y has had that value.

```
mon M = Always(x>0 -> Previously(y==x))
```

```
mon M = Always(x>0 ->
          let k=x in Previously(y==k))
```

```
mon M = Always(x>0 -> R(x))
min R(int k) = Previously(y==k))
```

y==k	k := x
y==x	x>0

Real-Time is Just Data

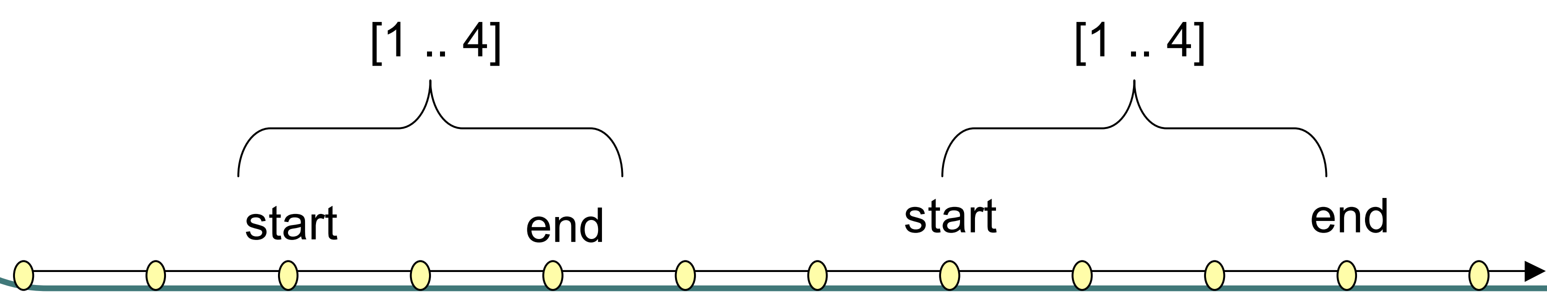
Property:
Every start is followed by an end within 1 to 4 seconds.

```
min WithinAbs(float t1, float t2, Term F) =
      clock <= t2 /\
      (F → t1 <= clock)  /\
      ( ~ F → @ EventuallyAt(t1, t2,F)) .

min Within(float t1, float t2, Term F)  =
      WithinAbs(t1+clock, t2+clock, F) .
```

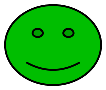
library

```
mon M = Always(start() => Within(1,4,end()))
```



Grammars

Property:
Locks are acquired and released nested.

lock lock release lock release release 

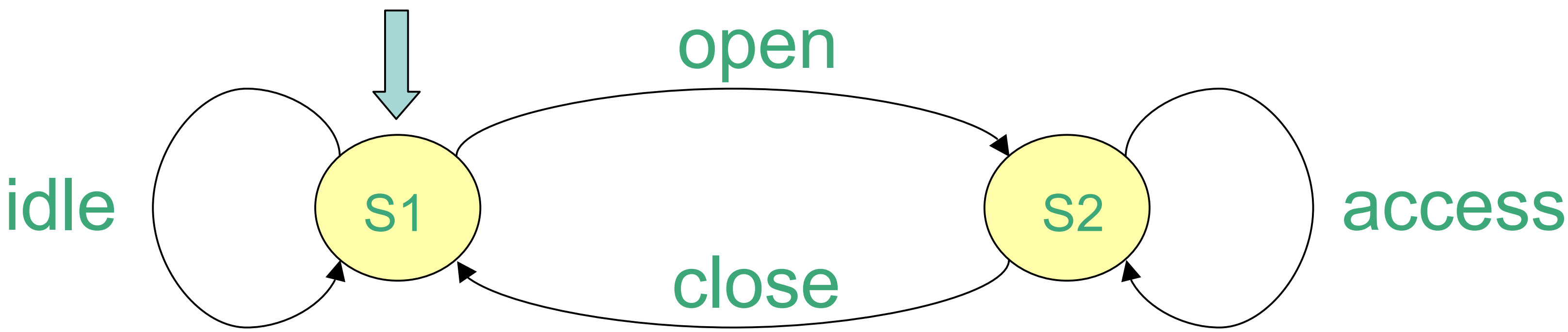
lock lock lock lock lock lock release 

```
mon M = Match(lock(),release())

min Match (Term l, Term r) =
   Empty() \/
   (l;Match(l,r);r;Match(l,r))
```

State Machines

Property:
File accesses are always enclosed by open and close operations.




```
max S1() = open -> @ S2()
        /\ idle -> @ S1()

min S2() = close  -> @ S1()
        /\ access -> @ S2()

mon M = S1()
```

Syntax

$$
\begin{aligned}
S &::= \mathbf{dec}\ D\ \mathbf{obs}\ O \\
D &::= R^* \\
O &::= M^* \\
R &::= \{\mathbf{max} \mid \mathbf{min}\}\ N(T_1\ x_1, \ldots, T_n\ x_n) = F \\
M &::= N = F \\
T &::= \mathbf{Form} \mid \textit{java primitive type} \\
F &::= \textit{java expression} \mid \mathbf{True} \mid \mathbf{False} \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \rightarrow F_2 \mid \\
&\qquad \bigcirc F \mid \odot F \mid F_1 \cdot F_2 \mid N(F_1, \ldots, F_n)
\end{aligned}
$$

Semantics

$$
\begin{array}{ll}
\sigma, i \models_D \mathit{exp} & \text{iff } \ 1 \leq i \leq |\sigma| \text{ and } \mathit{evaluate}(\mathit{exp})(\sigma(i)) == \mathit{true} \\
\sigma, i \models_D \underline{\text{true}} & \\
\sigma, i \not\models_D \underline{\text{false}} & \\
\sigma, i \models_D \neg F & \text{iff } \ \sigma, i \not\models_D F \\
\sigma, i \models_D F_1 \wedge F_2 & \text{iff } \ \sigma, i \models_D F_1 \text{ and } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \vee F_2 & \text{iff } \ \sigma, i \models_D F_1 \text{ or } \sigma, i \models_D F_2 \\
\sigma, i \models_D F_1 \rightarrow F_2 & \text{iff } \ \sigma, i \models_D F_1 \text{ implies } \sigma, i \models_D F_2 \\
\sigma, i \models_D \bigcirc F & \text{iff } \ i \leq |\sigma| \text{ and } \sigma, i+1 \models_D F \\
\sigma, i \models_D \odot F & \text{iff } \ 1 \leq i \text{ and } \sigma, i-1 \models_D F \\
\sigma, i \models_D F_1 \cdot F_2 & \text{iff } \ \exists j \text{ s.t. } i \leq j \leq |\sigma|+1 \text{ and } \sigma^{[1,j-1]}, i \models_D F_1 \text{ and } \sigma^{[j,|\sigma|]}, 1 \models_D F_2 \\
\sigma, i \models_D N(F_1, \ldots, F_m) & \text{iff } \left\{ \begin{array}{l}
\text{if } 1 \leq i \leq |\sigma| \text{ then:} \\
\quad \sigma, i \models_D F[x_1 \mapsto F_1, \ldots, x_m \mapsto F_m] \\
\quad \text{where } (N(T_1\ x_1, \ldots, T_m\ x_m) = F) \ \in D \\
\text{otherwise, if } i = 0 \text{ or } i = |\sigma| + 1 \text{ then:} \\
\quad \text{rule } N \text{ is defined as } \underline{\text{max}} \text{ in } D
\end{array} \right.
\end{array}
$$

How does it Work?

User defines these classes

e1 e2 e3 …

```
class Event {
  int x,y;
}
```

```
class Observer {
  Monitors mons;
  State state;
  eventHandler(Event e){
      state.update(e);
      mons.apply(state);
  }
}
```

```
class State {
  int x,y;
  update(Event e){
    x = e.x;  y = e.y; }
}
```

```
class Monitors {
  Formula M1, M2;
  apply(State s){
    M1.apply(s);
    M2.apply(s); }
}
```

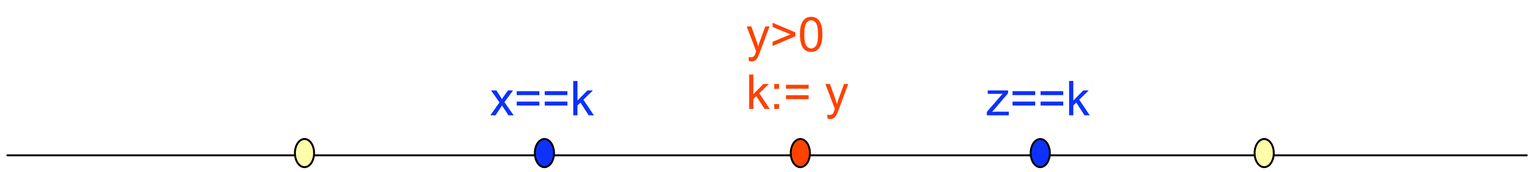
Implementation Ideas

- For each new event, the monitored formula is evaluated, resulting in either:
  - True - it is satisfied
  - False – it is violated
  - A non-reducable formula that the rest of the trace must satisfy.
- Then all formulas appearing after # (previous state) are evaluated and remembered for the next state.

Example Evaluation

```
mon M = Always(y>0 -> R(y))

min R(int k) = # k==x /\ @ k==z
```



Always(y>0 -> R(y))

R(int k) = # k==x /\ @ k==z

Evaluating

state	formula	k==x
x=77 y=0 z=18	Always(y>0 ->R(y))	k==77
x=28 y=77 z=2	Always(y>0 ->R(y)) /\ 77==z	k==28
x=1 y=0 z=77	Always(y>0 ->R(y))	k==1

Eagle for Java using AOP

- ```
  Observer Locks{
    var Thread t, FileSystem fs, int l;

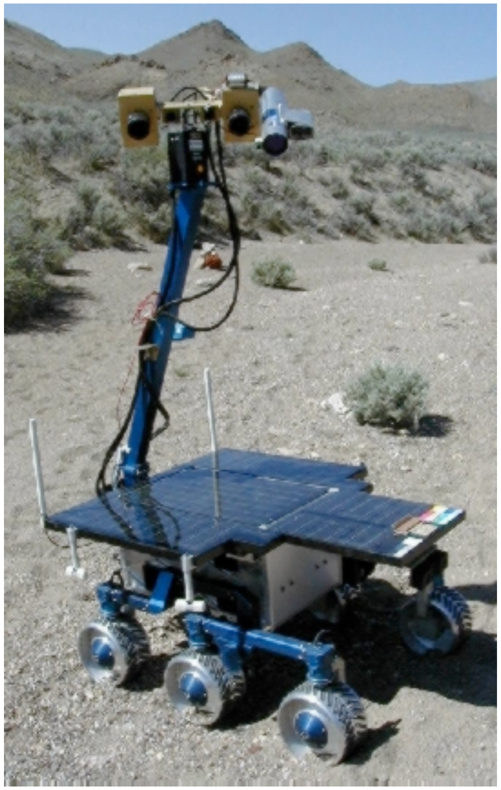
          mon M = Always(
                  [t?:fs?.lock(l?)]
                    Until(
                     ~<t:fs.lock(l)>true,
                      <t:fs.release(l)>true
                    )
              )
  }
  ```
  ```

But Properties are Hard to Formulate

To quote quite excellent NASA software engineer when asked what properties his system would have to satisfy:

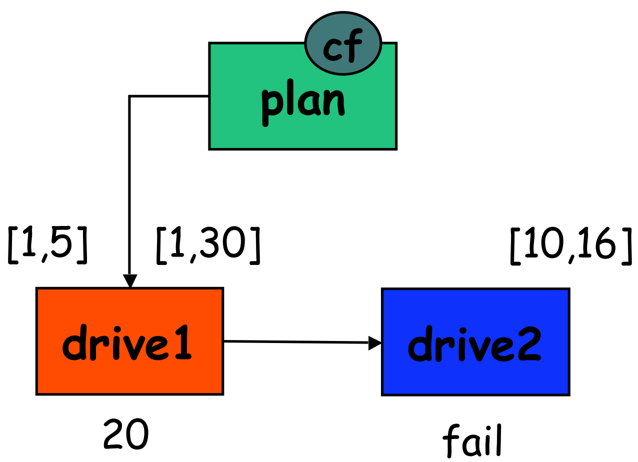
> "*I have absolutely no idea what properties this system should satisfy*".

K9 Planetary Rover Executive



- Executive receives plans from a planner for direct execution
- Plan is a hierarchical structure of actions
- Multi-threaded system (35K lines of C++)

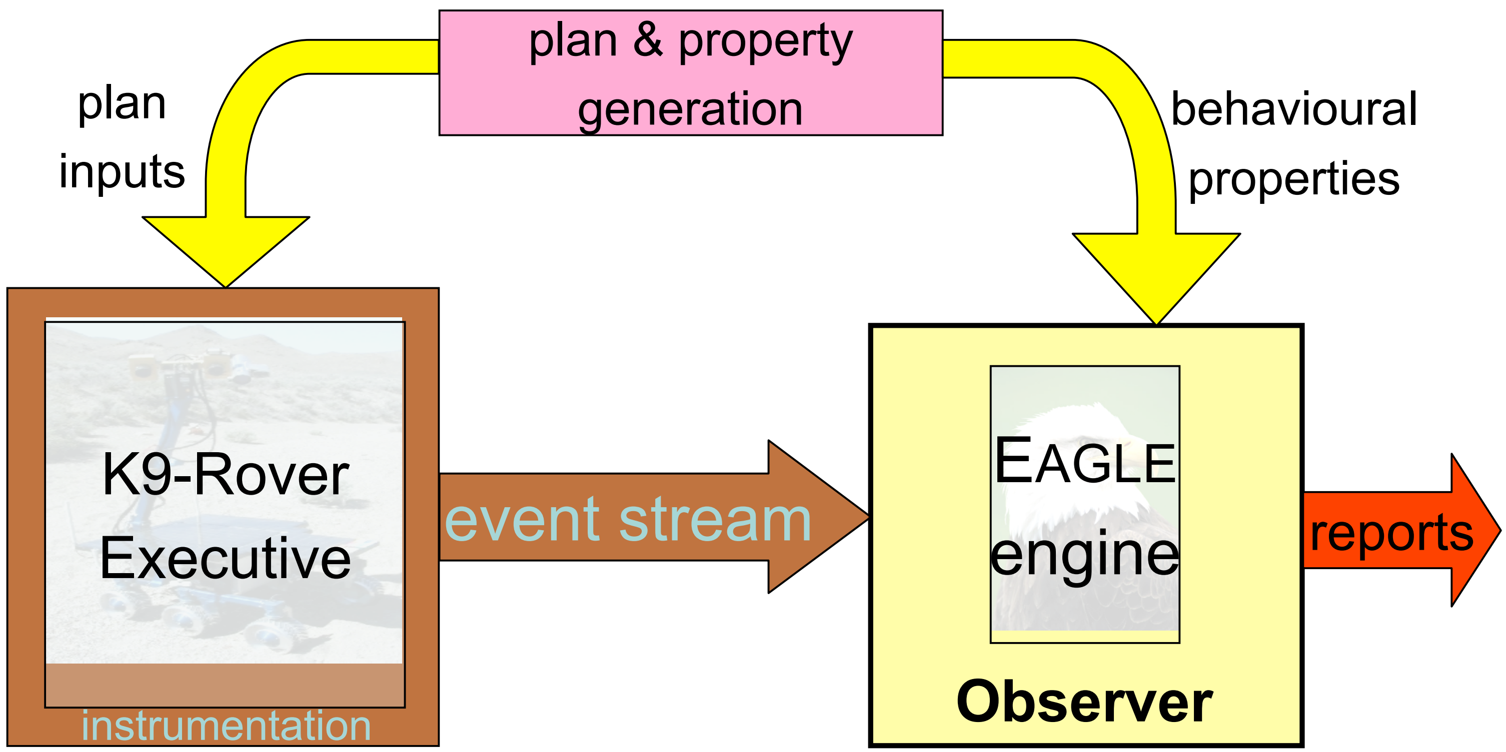
Example of Plan




```
(block :id plan
        :continue-on-failure
        :node-list (
            (task :id drive1
                :start-condition (time +1 +5)
                :end-condition (time +1 +30)
                :action BaseMove1)
            (task :id drive2
                :end-condition (time +10 +16)
                :action BaseMove2)
        )
)
```

With
Willem Visser and
Corina Pasareanu

Running X9



Runtime Verification Algorithms

- Requirement monitoring
  - The Eagle Temporal logic
- Concurrency Analysis
  - **Deadlock analysis**
  - Data race analysis
    - Low level data races
    - High level data races
    - Data flow races

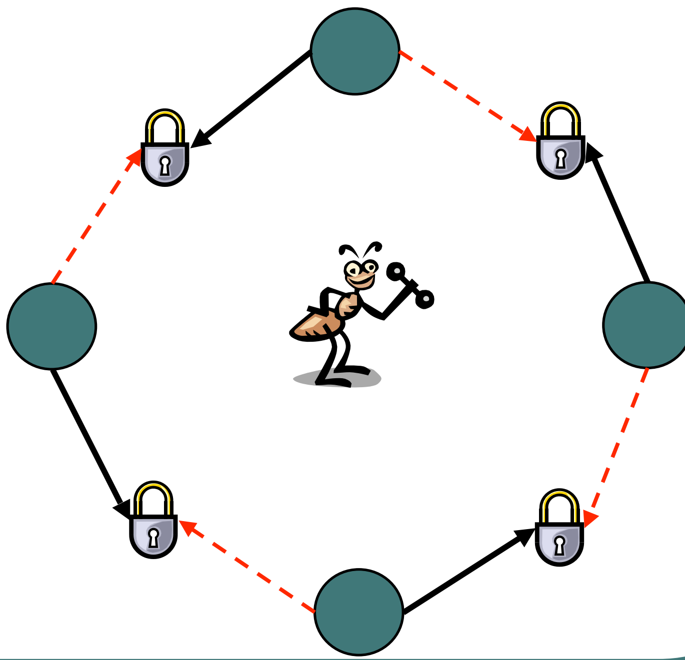
More Power – More Problems

Kestrel Tech.

- Multi-threaded programs may execute differently from one run to another due to the apparent randomness in the way threads are scheduled.
- Typically, testing cannot explore all schedules, so some bad schedules may never be discovered.

Cyclic Deadlocks

A resource deadlock can occur when two or more threads block each other in a cycle while trying to access synchronization locks (held by other threads) needed to continue their activities.

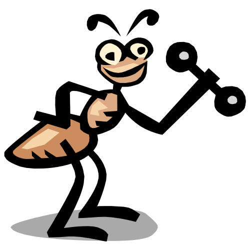


Cyclic Deadlocks

Deadlock: if T1 takes R1 and then T2 takes R2

T1:

```
lock(R1);
...
lock(R2);
...
release(R2);
...
release(R1);
```



T2:

```
lock(R2);
...
lock(R1);
...
release(R1);
...
release(R2);
```

Cycle Detection: A Simple Algorithm

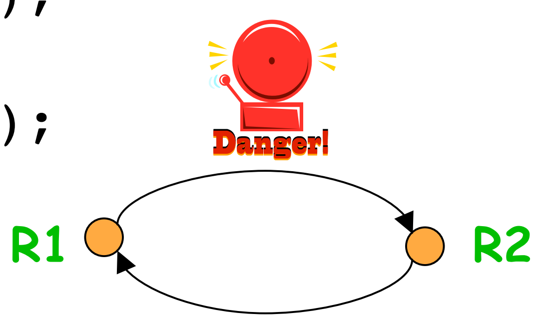
Deadlock: if T1 takes R1 and then T2 takes R2

T1:

```
lock(R1);
...
lock(R2);
...
release(R2);
...
release(R1);
```

T2:

```
lock(R2);
...
lock(R1);
...
release(R1);
...
release(R2);
```



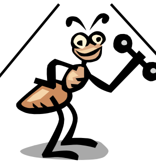
A Miracle?

- Deadlock potential detected even though a deadlock did not occur in that run
- Reason: we are checking a **stronger property:**
  - Weaker property: deadlock freedom
  - Stronger property: **cycle freedom**

Simple Algorithm Gives False Negatives

Guarded cycle

Deadlock cycle!



T1:

```
sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

T2:

```
sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

T3:

```
sync(L1){
  sync(L2){}
}
```

Thread segmented cycle

Singular cycle

4 deadlock potentials
Only one is real

Execution Trace

Trace

T1:

```
sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

T2:

```
sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

T3:

```
sync(L1){
  sync(L2){}
}
```

Event format:

l(<thread>,<lock>) - lock
u(<thread>,<lock>) - unlock
s(<thread>,<thread>) - start
j(<thread>,<thread>) - join

```
l(T1,G)
l(T1,L1)
l(T1,L2)
u(T1,L2)
u(T1,L1)
s(T1,T3)
l(T2,G)
l(T2,L2)
l(T2,L1)
u(T2,L1)
u(T2,L2)
u(T2,G)
l(T3,L1)
l(T3,L2)
u(T3,L2)
u(T3,L1)
j(T1,T3)
l(T1,L2)
l(T1,L1)
u(T1,L1)
u(T1,L2)
```

Full Algorithm

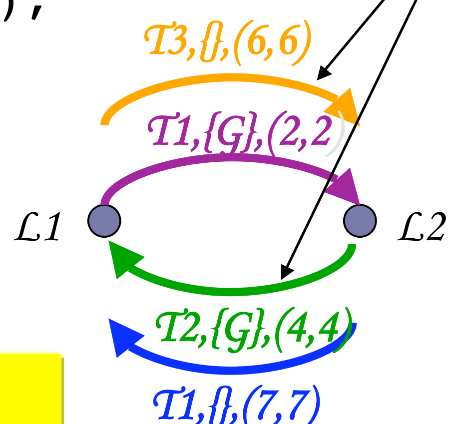
M:
```
new T1().start();
new T2().start();
```

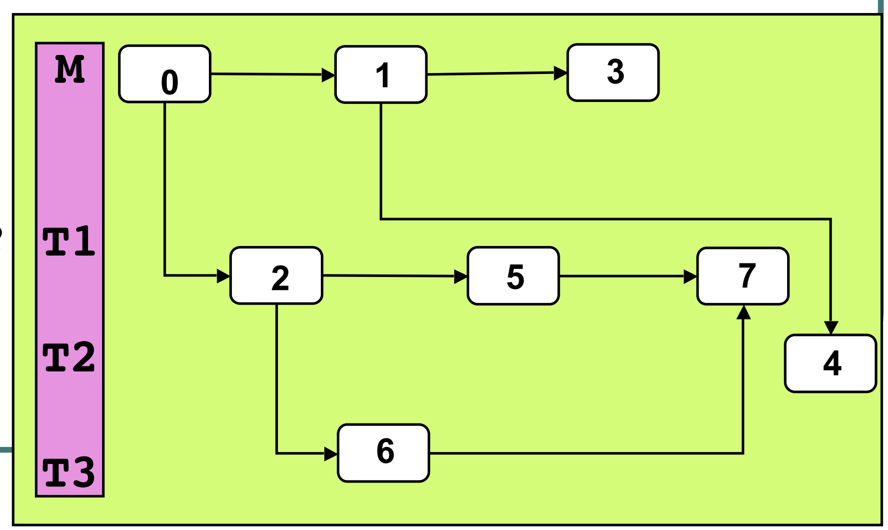
T1:
```
sync(G){
  sync(L1){
    sync(L2){}
  }
};
T3 = new T3();
j3.start();
J3.join();
sync(L2){
  sync(L1){}
}
```

T2:
```
sync(G){
  sync(L2){
    sync(L1){}
  }
}
```

T3:
```
sync(L1){
  sync(L2){}
}
```

One potential left, the real deadlock!





Valid Cycles:

1. *Threads:* must differ
2. *Guard sets:* must not overlap
3. *Segments:* must be parallel

Static Code Analysis Fails On Some Examples

```
class Main{
  Fork[] forks = new Fork[N];
  ..
  for(int i=0;i<N;i++){
    new Phisosopher(forks[i],
                    forks[(i+1)%N];
  };
}
```

Static analysis *cannot find this problem due to the dynamic creation of forks and the '%' operator (experiment with JLint).*

Model checking *works for N=20, but if program is deadlock free (introducing gate lock) N=3 is max using 3 minutes (JPF).*

Runtime Verification Algorithms

- Requirement monitoring
  - The Eagle Temporal logic
- Concurrency Analysis
  - Deadlock analysis
  - Data race analysis
    - **Low level data races**
    - High level data races
    - Data flow races

Data Races

Standard definition:

A data race occurs when two concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the accesses from being simultaneous.

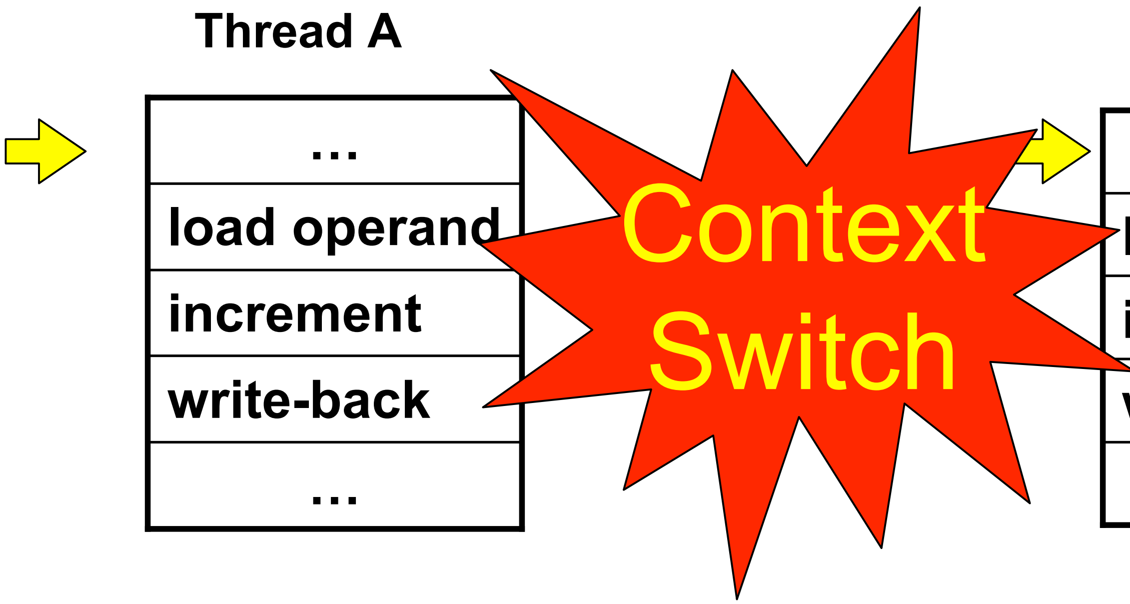
A Classic Java Example

Let’s consider the function increase(), which is a part of a class that acts as a counter

```
public void
increase()
{
        counter++;
}
```

Although written as a single “increment” operation, the “++” operator is actually mapped into three JVM instructions [load operand, increment, write-back]

Example – Continued



counter = 4

We shall refer to this traditional notion of data race as a *low-level data race*, since it focuses on a single variable

Low-Level Data Races

- The standard way to avoid low-level data races on a variable is to protect the variable with a lock: all accessing threads must acquire this lock before accessing the variable, and release it again after.
- There exist several algorithms for analyzing multi-threaded programs for low-level data races.
- We will mention the Eraser algorithm here (Savage et al 97).

Simple Algorithm

```
T1:

synchronized(R1){
  sum = sum + 100;
}
```

```
T2:

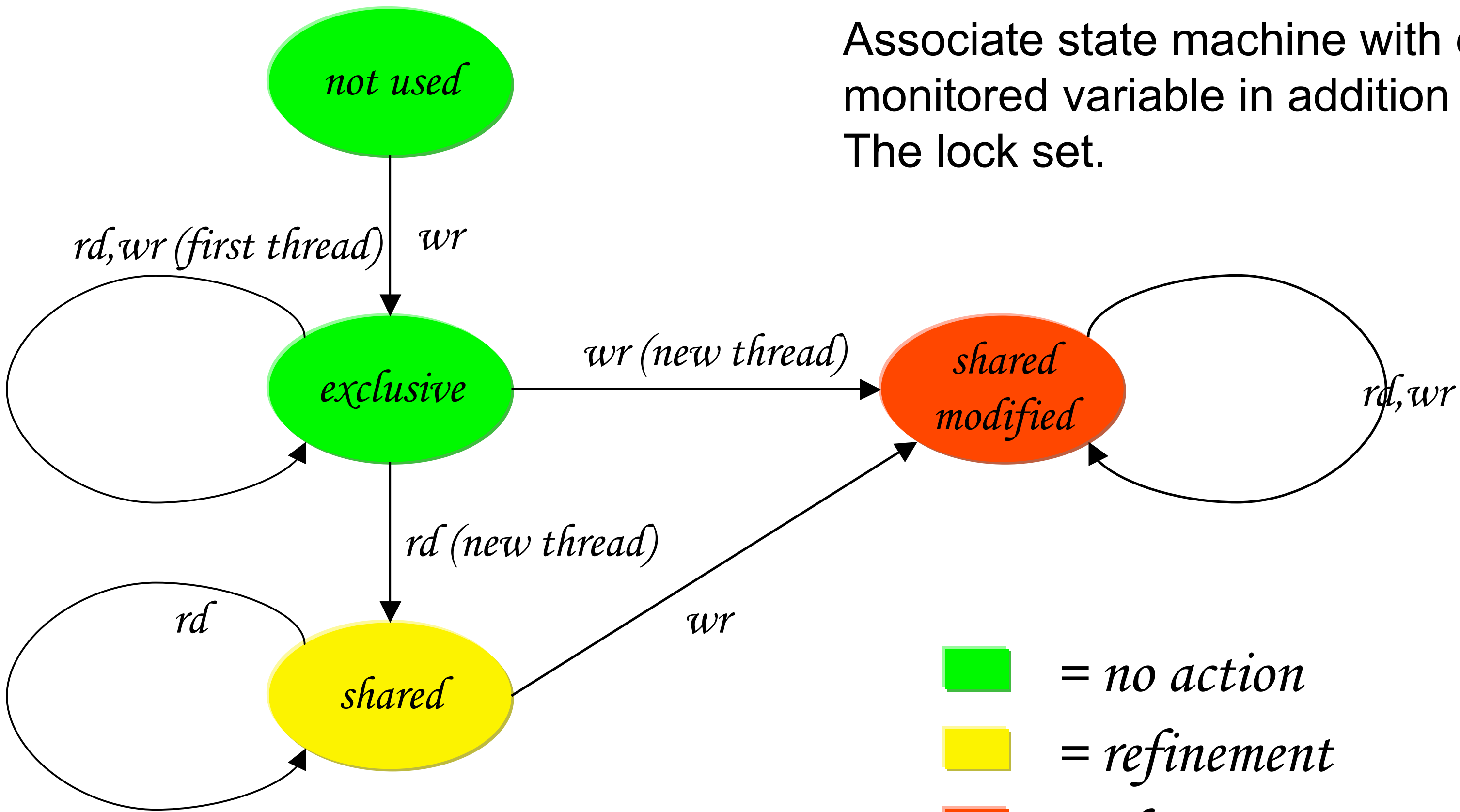
synchronized(R2){
  sum = sum + 50;
}
```

Initially: Lockset = {}

T1 executes: Lockset = {R1}

T2 executes: Lockset = Lockset $\cap$ *{R2} = {}*

Full Algorithm Reduces False Positive



Associate state machine with each monitored variable in addition to The lock set.

- = *no action*
- = *refinement*
- = *also warnings*

Runtime Verification Algorithms

Kestrel Tech.

- Requirement monitoring
  - The Eagle Temporal logic
- Concurrency Analysis
  - Deadlock analysis
  - Data race analysis
    - Low level data races
    - **High level data races**
    - Data flow races

Data Race

```
void swap() {
  lx = c.x;
  ly = c.y;
  c.x = ly;
  c.y = lx;
}
```

```
void reset() {
  synchronized(this){
    c.x = 0;
  }
  synchronized(this){
    c.y = 0;
```

Pair of coordinates *x and y*.

Two threads.
Problem: thread order non-deterministic.
Data corruption possible!

Lock protection needed.

High-Level Data Race

5,8

```
void swap() {
  synchronized(this){
    lx = c.x;
    ly = c.y;
  }
  synchronized(this){
    c.x = ly;
    c.y = lx;
  }  8,0
}
```

```
void reset() {
  synchronized(this){
    c.x = 0;
  } 0,8
  synchronized(this){
    c.y = 0;
  }
}  8,0
```

Result is neither a swap or a reset

All field accesses synchronized: Eraser reports no errors.
No classical data race for these threads, but clearly undesired behavior!
Problem: **swap** may run while **reset** is in progress!

The Problem

- The **reset** method releases its lock in between setting x and then setting y.
- This gives the **swap** method the chance to interleave the two partial resets.
- The **swap** method “has it right”: it holds its lock during operation on x and y.
- **This difference in views can be detected dynamically**.
- Depends on at least one thread getting it right.

The Solution

- This difference in views can be detected dynamically.
- Essentially, this approach tries to infer what the developer intended when writing the multi-threaded code, by discovering view inconsistencies.
- Depends on at least one thread getting it right.

Views express per thread
what fields are guarded by a lock.

Some Examples of Views:

Thread a

```
synchronized(L)
{
    access(x);
    access(y);
}
```

{ {x,y} }

Thread b

```
synchronized(L)
{
    access(x);
}
synchronized(L)
{
    access(y);
}
```

{ {x},{y} }

Thread c

```
synchronized(L)
{
    access(x);
}
synchronized(L)
{
    access(x);
    access(y);
}
```

{ {x},{x,y} }

Thread d

```
synchronized(L)
{
    access(x);
}
```

{ {x} }

Consistent:	a and c, a and d
Inconsistent:	a and b

The Algorithm

1) For each thread, for each lock, identify all fields covered by
that lock (views).

2) For each thread, find the views that have no other view
that contains them (maximal views).

3) For each pair of threads t1 and t2: find the intersection
between t1's maximal view and the views of t2.

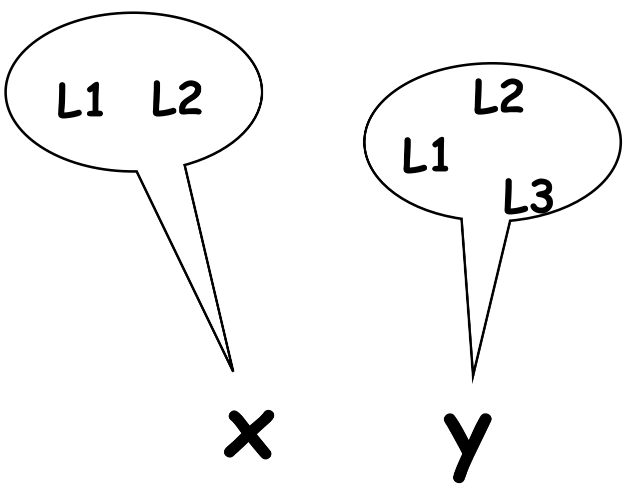
4) Verify that those intersections form a chain. That is:

$$s1 \subseteq s2 \subseteq s3 \subseteq \cdots$$

Low-Level versus High-Level Data races

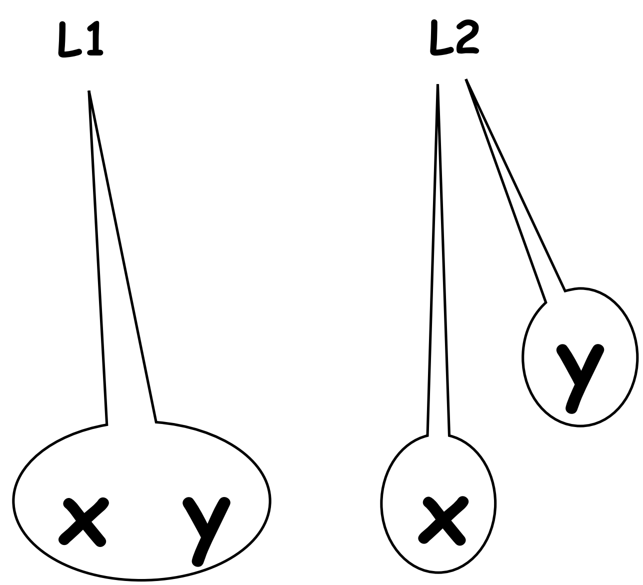
Low-Level

For each variable: a lock set



High-Level

For each lock: a variable set (several)



Applying Algorithm to Example

```
void swap() {
  synchronized(this){
    lx = c.x;
    ly = c.y;
  }
  synchronized(this){
    c.x = ly;
    c.y = lx;
  }
}
```

x , y

maximal of swap

```
void reset() {
  synchronized(this){
    c.x = 0;
  }
  synchronized(this){
    c.y = 0;
  }
}
```

x

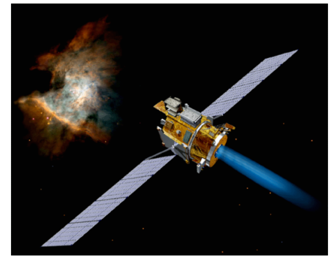
y

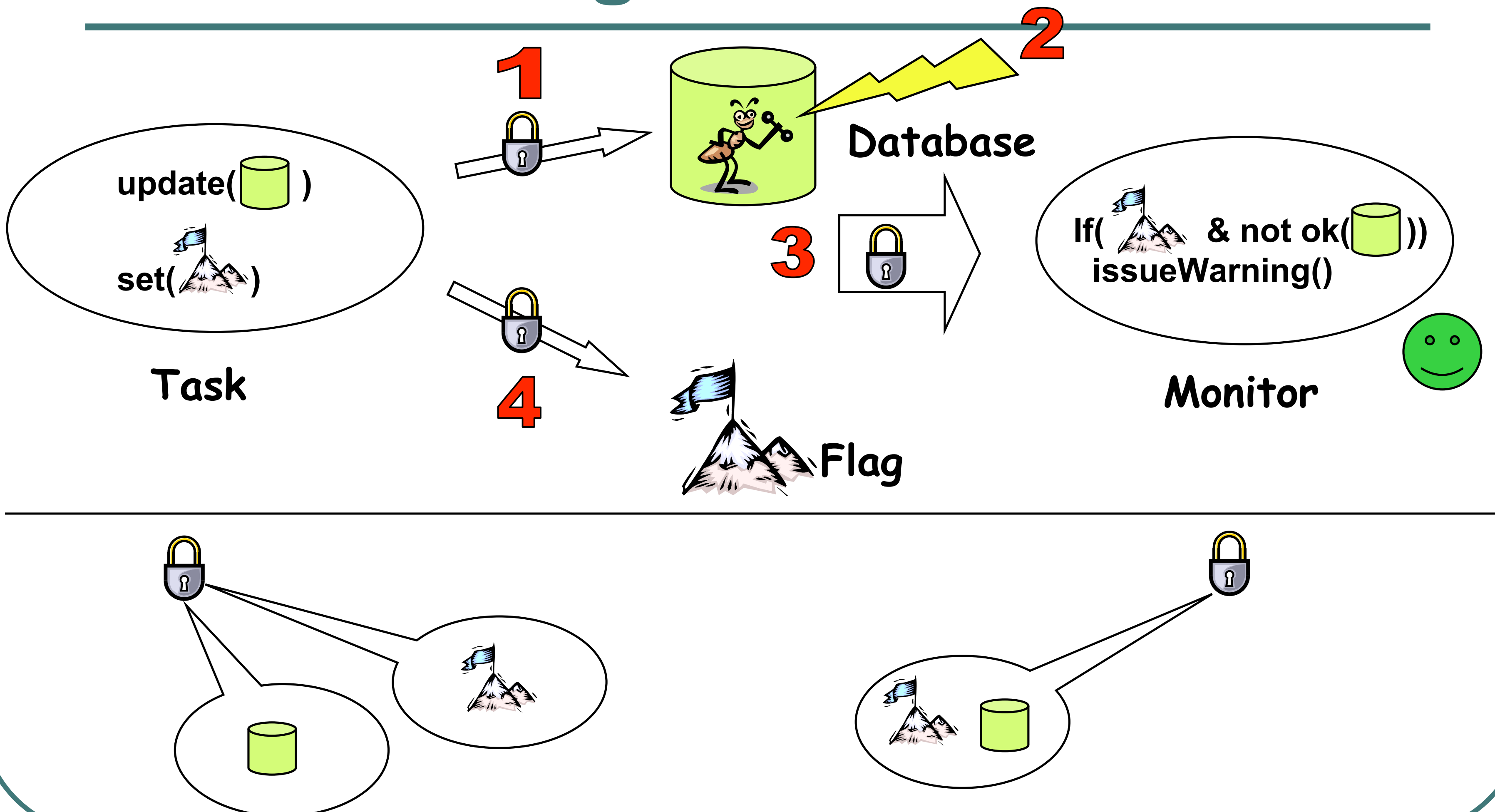
Overlaps are: {x} and {y}.

$\{x\} \not\subseteq \{y\}$

$\{y\} \not\subseteq \{x\}$

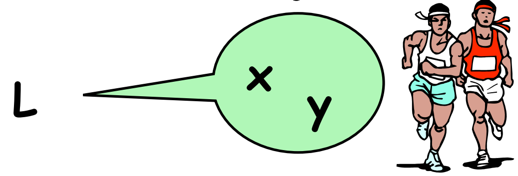
HL Data Race in Remote Agent

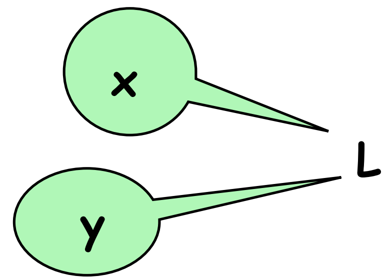




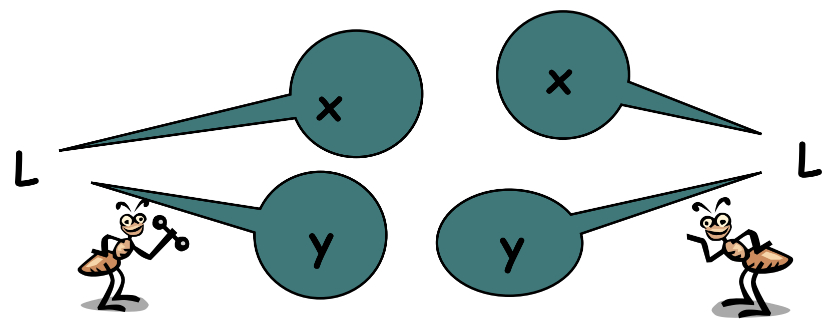
Neither Sound Nor Complete

False positive when one thread uses coarser locking that required due to efficiency.



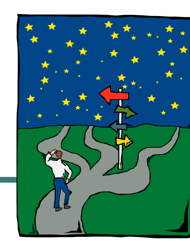


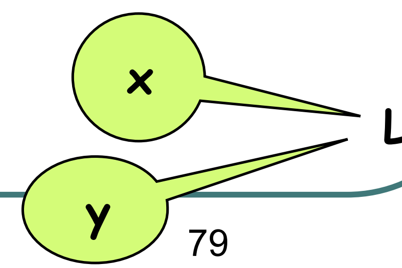
False negatives when:



- All threads use the same locking

- Random execution trace does not expose problem





So, what is it good for?

- Much higher chance of detecting an error than if one relies on actually executing the particular interleaving that leads to an error, without requiring much computational resources.
- Developers seem to follow the guideline of view consistency to a surprisingly large extent.

Runtime Verification Algorithms

- Requirement monitoring
  - The Eagle Temporal logic
- Concurrency Analysis
  - Deadlock analysis
  - Data race analysis
    - Low level data races
    - High level data races
    - **Data flow races**

Recall The High Level Data Race

```
void swap() {
  synchronized(this){
    lx = c.x;
    ly = c.y;
  }
  synchronized(this){
    c.x = ly;
    c.y = lx;
  }
}
```

```
void reset() {
  synchronized(this){
    c.x = 0;
  }
  synchronized(this){
    c.y = 0;
  }
}
```

Problem: **swap** may run while **reset** is in progress!

Repairing the Situation: Making reset Atomic

5,8

```
void swap() {
  synchronized(this){
    lx = c.x;
    ly = c.y;
  }  5,8
  synchronized(this){
    c.x = ly;
    c.y = lx;
  }
}  8,5
```

```
void reset() {
  synchronized(this){
    c.x = 0;
    c.y = 0;
  }  0,0
}
```

Reset invoked after swap, but has no effect

Problem:
- **reset** may run while **swap** is in progress!
- **swap** then continues operating on outdated values

The Problem: Data Flow Across Synchronized Blocks

```
void swap() {
  synchronized(this){
    lx = c.x;
    ly = c.y;
  }
  synchronized(this){
    c.x = ly;
    c.y = lx;
  }
}
```

lx and **ly** defined!
store values locally

lx and **ly** used!
may be outdated

Shared data "escape" beyond first synchronized block!

Algorithm checks whether shared data escape synchronized blocks.

Algorithm

- Enumerate synchronized blocks.
- Mark values as shared or unshared.
- Mark local variables with
  - the identity of synchronization block where defined.
  - Whether they contain a shared variable.
- For each use of a local variable, check:

  block where used = block where defined.

Determining Sharedness

If instruction **creates** stack elements (getfield, method call)

- if inside a **synchronized block:** stack elements generated are marked as shared
- else: stack elements generated are marked as local

Determining Sharedness of Return Values of Methods

```
synchronized(this){
  lx = c.getX();
}
```

Method call inside synchronization:
return value is **shared**

```
synchronized int getX() {
  return x;
}
lx = c.getX();
```

Method call outside synchronization:
callee uses synchronization:
return value is **shared**

```
int getX() {
  return x;
}
lx = c.getX();
```

Method call outside synchronization:
no synchronization in callee:
return value is **local**

Workshop

Fifth International Workshop on

Runtime Verification

RV'05

CAV'05
June, 2005
Edinburgh
Scotland